Tractable Goal Selection with Oversubscribed Resources

Gregg Rabideau, Steve Chien, David McLaren

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Dr, Pasadena, CA 91109
firstname.lastname@jpl.nasa.gov

Abstract

We describe an efficient, online goal selection algorithm and its use for selecting goals at runtime. Our focus is on the re-planning that must be performed in a timely manner on the embedded system where computational resources are limited. In particular, our algorithm generates near optimal solutions to problems with fully specified goal requests that oversubscribe available resources but have no temporal flexibility. By using a fast, incremental algorithm, goal selection can be postponed in a "just-in-time" fashion allowing requests to be changed or added at the last minute. This enables shorter response cycles and greater autonomy for the system under control.

Introduction

Consider the following properties of many autonomous systems:

- embedded computing resources are typically scarce
- response times can be critical

In other words, our algorithms must be fast. In particular, for time-critical problems, we must be able to make guarantees on responsiveness. Also consider the following:

- the time horizon is bounded (i.e. the autonomous system does not need to be indefinitely autonomous)
- some parts of the planning problem (ones that can be well predicted) can be solved in advance

This suggests that, in many cases, we are not trying to solve the general "planning problem". For example, active spacecraft rarely operate for more than two weeks without ground communication. Therefore, onboard plans do not need to cover a large time frame. Also, parts of the planning problem can be solved well in advance (e.g. spacecraft orbit predictions) and pre-compiled solutions can be utilized by the onboard planner. Limiting the scope of the problem gives us some hope at finding efficient solutions. Our work focuses on a restricted planning problem with a tractable solution that has a guaranteed worst-case complexity.

Specifically, we address the problem of providing high-level, goal-based autonomy for computationally limited robotic systems. We enable on-board and remote goal triggering through the use of an embedded, dynamic goal set that can oversubscribe resources. From the set of conflicting goals, a subset must be selected that maximizes a given quality metric.

Our approach solves the following problem:

- Goals have fixed start times and durations (making this a goal selection problem rather than an NP-hard scheduling problem).
- Goals can have flexible sub-activities for execution but the selection of alternative branches or timings cannot require search (e.g. we may wait for an event, retry a fixed number of times, or branch based on a condition but no backtracking occurs). This can be viewed as backtrack-free hierarchical task network planning.
- Goals can conflict by exceeding the limits of shared resources (e.g. oversubscription) with selection based on a strict priority ordering (i.e. a goal can never be pre-empted by any number of lower priority goals).
- Goals can be added, removed, or updated at any time, and the "best" goals will be selected for execution.

While these algorithms are general, we have implemented them as prototype extensions to the Virtual Machine Language (VML) [Grasso 2008] execution system. VML is advanced, multi-mission flight and ground software developed for NASA flown on a number of past and current missions including: The Spitzer Space Telescope, Mars Odyssey, Stardust, Genesis, Mars Reconnaissance Orbiter, Phoenix, and Dawn. The prototype goal and resource concepts are added to the language for both planning and execution purposes. Specifically, the Goal Manager (GM) maintains the set of requested goals, their priorities, and their interactions (i.e. shared resource constraints). When a goal is submitted, the GM quickly analyses the goal to determine whether or not it should be selected for execution. When the current time approaches the scheduled start time of a selected goal, the goal is committed and satisfied by spawning the corresponding VML sequences.

Our work is motivated by experience from space mission operations, such as autonomous operations of the Earth Observing One satellite [GSFC] and operations conducted by the Autonomous Sciencecraft (ASE) flight and ground software [Chien et al. 2005]. In ASE, events are detected on-board which trigger changes in goal requests. For example, images taken of the Earth can be processed on-board to detect interesting events such as

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

volcanic eruptions. These detections can then trigger changes to upcoming goals such as increasing the priority of requests for images of the same volcano. On the ground, sensorweb processing may detect similar events and upload new goal requests in a short time frame.

We demonstrate our prototype implementation on these scenarios. In the Goal Manager (GM), goal selection is postponed until the latest possible time, allowing goals to be added, removed or changed just prior to execution. This dynamic goal set enables additional autonomy capabilities such as on-board and ground-based event triggering, similar to ASE. The GM, however, does not require a full planner and has theoretical guarantees on worst-case response time. In these scenarios, start times of goals are assumed to be fixed. This is a reasonable assumption due to the nature of a spacecraft in orbit – opportunities for communications and science observations occur at specific (repeating) times. Also, we have found that many spacecraft resource constraints can be abstracted to the goal level. For example, the EO-1 spacecraft can point science instruments to only one target at a time. Thus, for target locations in close proximity, we must choose one of possibly many observation goals. We take advantage of these assumptions to develop efficient algorithms that provide advanced onboard autonomy capabilities.

Finally, we build on previous work [Rabideau and Chien, 2008] by providing an empirical analysis that supports the theoretical results. In these experiments, the GM runs in polynomial time, generating schedules that rank near optimal.

Resource Constraints

The primary driver for goal selection comes from the constraints on resources shared by the goals. We define a resource constraint as a value and a bound on that value over a period of time. Resource constraints can exist as part of goals, activities, or sequences. A combination of effects of constraints on the same resource conceptually comprises a timeline (although we do not maintain an explicit representation of a timeline). Resource constraints have the following attributes:

```
ResourceConstraint
{
  IdType          id;
  ResourceType    type;
  TimeType        start;
  TimeType        end;
  ValueType       value;
  ValueType       initial;
  ValueType       min;
  ValueType       max;
}
```

The `id` uniquely identifies the affected resource for the purpose of analyzing the interaction with other resource constraints. The `type` specifies the type of effect that the constraint has on the resource. The time range specifies the temporal scope of the constraint. The last four values specify, respectively: the constraint value, the initial value of the resource in the absence of all constraints, the minimum valid resource value, and the maximum valid resource value.

There are four fundamental types of resource constraints: `Producer`, `Consumer`, `Assigner`, and `Requirement`. A `Producer` adds the constraint value to the resource at the start of the time range and subtracts it at the end (where the end may be infinity). A `Consumer` subtracts at the start and adds at the end. An `Assigner` simply assigns the constraint value at a specific time point. A `Requirement` specifies only a constraint on the value of the resource over a period of time (i.e. it has no effect on the resource value).

A resource constraint can be defined for any type of value as long as the following set of operators is available: `+`, `-`, =, <, and `==`. The arithmetic operators allow us to compute resource values from a set of interacting resource constraints, and the boolean operators are for testing the validity of computed resource values. For example, for a single producer, we would add the produced value to the initial value and compare the result to the maximum value. If the constraint check fails, the resource value is considered invalid (i.e. has conflicting constraints).

We have demonstrated four common types of resource values: `int`, `double`, `string`, `set<string>`. The definitions of the operators are intuitive for simple types such as integers, doubles, and strings. For sets, we define them as follows: addition (+) is set union, subtraction (-) is set subtraction, assignment (=) replaces all values in one set with values from another, less (<) is a lexicographical ordering on two sets, and equals (==) returns true if each element in one set is equal to exactly one element in the other set.

For set resources, we introduce another operator for set containment. This allows us to specify a constraint that requires the computed resource value (which is a set of values) to contain the constraint value.

Goals

We define a goal to represent the request for execution of an activity or set of activities. Goals have the following attributes:

```
Goal
{
  IdType                   id;
  PriorityType             priority;
  TimeType                 start;
  TimeType                 end;
  set<ResourceConstraint> constraints;
}
```

The `id` is used to uniquely identify the goal. The goal `priority` is used to rank goals. The `start` and `end` values specify the expected temporal scope of the goal. Due to system uncertainties at the time the goal is requested, the start and end times contain only the requested or expected values. Goals also maintain a set of resource constraints that must hold for the goal to execute. Similar to the start and end times, resource constraints contain the requested or expected values for the resources.

The goal attributes are used for selecting and dispatching goals for execution. In addition, a goal must specify what is to be done when it is dispatched. Typically, this involves spawning a sequence to start execution at a given time. Essentially, we define goals as a summary of the intent and effects of one or more sequences.

Example: Onboard File System

When defining goals and resources, we worked to find a balance between a representation that is general and powerful, but also has the details required for efficient resource analysis and goal selection. We show the power of the representation with an example: managing an onboard file system. This example is of particular interest to us because many spacecraft (including EO-1) must deal with data products stored on an onboard file system. Typically, science activities write data to a file while engineering procedures read and downlink files, deleting them when appropriate to free up space.

We can model this file system with five goals and four resources. The goals represent file system requests: create, delete, read, write, and format. The resources shared by these goals are: a set of file handles (100), limited disk space (1024K), a current directory listing, and finally an exclusive use of the file system by certain operations.

Each goal instance creates and adds the resource constraints for that goal type. Creating a file consumes one of the 100 available file handles, and produces a file with the specified unique ID. Deleting a file produces a file handle while consuming the file with the specified unique ID. It also produces disk space equal to the size of the file. Writing to a file consumes available memory equal to the size of the data written. Both writing and reading require that the unique file ID is a member of the set of available file IDs, and that no other reads or writes occur at the same time.

Each goal type also defines the required method for executing the goal. For example, creating a file would call `fopen()` on a Unix operating system.

Goal Selection Algorithm

We present an algorithm for selecting goals with oversubscribed resources. The pseudo-code for the core of the algorithm (updating the set of selected goals) is shown in Figure 1. The algorithm can be categorized as a repair-based approach with no search – the constraints and priorities define exactly which goals to choose. We focus on the re-selection that is required when the requested goal set changes, either by adding a new goal or removing an existing goal[1]. While making selections, goal parameter values (e.g. start times) are assumed to be fixed. The result is a set of non-conflicting "best" goals that have been selected for execution. After selections are made, conflicting goals are retained for additional consideration in the event of future changes to the goal set. In this implementation, conflicts are defined by shared resource interactions, and choices are made among conflicting goals using a strict priority rule. Only the highest priority goals are selected, with ties broken by earliest request time (i.e. first-come-first-served).

```
void updateSelectedGoals(g)
  for each Goal $g_s$ in selectedGoals
      with priority <= priority of g
    remove $g_s$ from selectedGoals
  for each Goal $g_i$ in allGoals
      with priority <= priority of g
      in decreasing priority order
    if wasStarted($g_i$)
      or isBestGoal($g_i$, selectedGoals)
      add $g_i$ to selectedGoals

bool isBestGoal(g, selectedGoals)
  for each interacting Goal $g_i$ of g
    if $g_i$ is in selectedGoals
       and $g_i$ conflicts with g
      return false
  return true
```

Figure 1: Incrementally updating the set of selected goals

The Goal Manager (GM) maintains data structures that enable efficient goal selection and dispatch. One set of goals is sorted by priority so that the algorithm can efficiently perform goal selection. The goals are also sorted by start time so that it can quickly find the next selected goal to dispatch for execution. For each goal, the GM maintains a set of interacting goals (i.e. goals that share a resource) to assist resource reasoning. According to the priority rule, a goal will be selected and dispatched for execution if and only if it does not conflict with a higher priority goal, which does not conflict with an even higher priority goal, etc. For example, in Figure 2 we assume that overlapping goals conflict. Goal C is highest priority and is selected. Goal B is not selected because it conflicts with C. Because B is not selected, goal A is selected even though it is lower-priority than B. Therefore, goals A and C are the

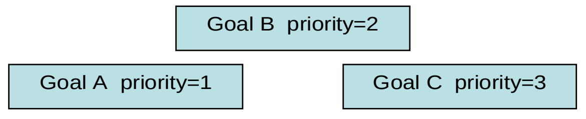

Figure 2: Goals A and C are selected

[1]Changing parameters of a goal (e.g. start time, priority) can be implemented by removing the goal and adding it back with new values.

"best" goals.

Adding and removing goals involves three steps: updating the sets of interacting goals, updating the sorted goal sets, and updating the selected goal set. This last step is the most interesting and is shown in pseudo-code in Figure 1. In this step, we first de-conflict the schedule by removing all goals with lower priority than the goal being added or removed (lines 2-4). Higher priority goals are unaffected and can remain selected. Next, we re-evaluate each of the lower-priority goals (lines 5-9). Evaluating a goal `g` involves adding the resource effects of `g` to the effects of all selected (i.e. higher-priority) goals that interact with `g` (lines 13-18). If the combined resource effects are invalid, then `g` will not be selected (lines 19-20). Finally, dispatching goals within a given time range simply involves finding selected goals that fall in that range. The dispatch function is intended to be called periodically with a small time range covering the near future.

It is important to note that the low-priority conflicting goals are retained so that changes to the goal set can be made at any time (goals added, removed, or updated), and goals will be dispatched from the latest set of selected goals. Once a goal has started executing (determined by the "wasStarted" function on line 8) it will thereafter be selected regardless of priority. A goal expires if it is unselected and falls in the past, or if it is selected and all of its interacting goals completely fall in the past. Expired goals are periodically removed from the goal sets. As a final note, the definition of "interacting" and "conflicting" can be arbitrary boolean operators. In this work, resources define which goals interact and resource calculations are performed to determine conflicts.

Algorithm Analysis

We now describe the run-time computational complexity of our goal selection algorithms. Selecting the best goals and updating the cache (lines 21-39) is:

$$O(MlgM + N(lgM + X_i(lgM + S_i)))$$

where N is the number of all goals, M is the number of selected goals, X_i is the number of interacting goals for goal i, and S_i is the number of resources shared by goal i. Goals are stored in tree data structures with a log-based lookup. The first term comes from removing lower-priority goals from the selected goals (lines 22-24). The longer second term comes from re-selecting the best goals (lines 25-39).

Assuming worst case, where each goal interacts with every other goal ($X_i == N$ for all i), each goal uses all resources ($S_i == R$ for all i), and all goals are selected ($M == N$):

$$O(NlgN + N(lgN + N(lgN + R)))$$

Or:

$$O(NlgN + NlgN + N^2lgN + N^2R)$$

Since N^2lgN dominates $NlgN$:

$$O(N^2lgN + N^2R)$$

And assuming that R is constant (defined by the domain), we have:

$$O(N^2lgN)$$

This is a theoretical worst case complexity. In practice, each goal will typically use a subset of the resources, and many of the goals will not be selected for execution. More importantly, goals interact with a small number of other goals (X_i is constant) due to the temporal scope of the resource constraints (i.e. effects on resources have limited extent). This gives us:

$$\Theta(NlgN)$$

Our empirical analysis (discussed in a later section) supports this average case bound.

Once we have cached the best goals, checking a specific goal is a simple lookup in the set. Dispatching a selected goal for execution is:

$$O(lgN + lgM)$$

The first term is from the lookup for goals due for execution which we assume to be small (typically one). The second term is from the lookup in the set of selected goals. Assuming worst case where all goals are selected ($M == N$), we get:

$$O(lgN)$$

Finally, we take a look at the expected quality of the output of the algorithm. Our primary claim is that the algorithm is optimal for the given priority rule. In other words, a goal with priority P will always be selected in place of any number of goals with priority less than P. Intuitively, this follows from the decreasing priority order in which goals are selected. Now consider scoring the selected goal set with a weighted sum using weights sufficiently large at priority P to outweigh all goals at priority less than P. Our goal selection algorithm will maximize this score, but only when the requested goals are assigned unique priorities. If goal priorities are not unique, the overall weighted sum of priorities depends on the order in which we select goals with equal priority. In our implementation, the goal submitted first is selected first. This goal, however, may use more resources than the other goals with the same priority, accommodating fewer goals at lower priorities, and producing an overall lower score. Our initial empirical analysis, however, shows that our solutions do not fall far from the maximum score.

Algorithm Assumptions, Limitations, and Requirements

We make several assumptions to keep the goal selection algorithm simple and efficient.

- We do not solve the general planning problem. We only decide which high level goals should be selected. We do not search for alternate methods of achieving the high level goals. While less powerful, this tends to be more accepted by spacecraft engineers who prefer consistency and predictability. The tasks of goal decomposition and command execution are left to an executive or sequencing engine (e.g. VML). These systems can be very expressive and allow goals to be expanded in a complex, context-dependent manner.
- We do not solve the general scheduling problem. We only decide on *which* subset of requested goals and activities to add to the plan, not on *when* they should be scheduled. Goals and activities must be submitted with predetermined start times. As an example, for an orbiting spacecraft with repeating science opportunities, this restricted form of planning can select which observation to perform on a specific orbit, but can not select alternate overflights for a particular observation.
- We are only reasoning at the goal level. Resource reasoning is performed on goal resources which are assumed to be abstractions of the expected use of resources by the lower-level commands. We found this abstraction useful for many of the EO-1 resource constraints
- We also assume that goals are ranked using the strict priority rule. In other words, any number of low-priority goals can be preempted by a high-priority goal. When goals have equal priority, the goal that was requested first will take precedence (i.e. first-come-first-served). EO-1 scientists were most comfortable with this simple priority scheme.

It is worth pointing out that even with these restrictive assumptions, the goal replacement capabilities we are offering far exceed what is available on typical spacecraft today, whether implemented in general commanding capabilities or custom flight software.

To benefit from these capabilities, however, users must encode some additional knowledge when defining goals compared to defining activities or sequences strictly for the purpose of execution. First, users must provide some form of selection criteria. In our case, this is a priority for the goal. The user must also specify a summary of the expected resource usage for each goal. Where resource use at runtime may be intricate or even implicit, goal resources force the user to define resource use in an explicit and predictable way. Finally, users must provide an expected start and end time for the activity or sequence requested by the goal. This is necessary for predicting the temporal scope of the resource use.

Planning and Execution with VML

Designed as a multi-mission application, VML is one of the most advanced onboard execution systems in widespread use for NASA missions [Grasso 2004]. Missions currently using VML include Odyssey, Spitzer, Dawn, MRO, and Phoenix. On these missions, VML has been used for a wide range of sequencing functions including: launch routines, orbit insertion, entry-descent-and-landing, science acquisition, and fault response.

We have implemented goals, resources, and the goal selection algorithm as prototype extensions to VML. A new thread/task, the Goal Manager (GM), implements the goal selection algorithm and invokes the dispatch function periodically. Finally, new user interface functions are added to allow goals to be added, removed, or changed.

At runtime, we ultimately need a set of executable commands that achieve the selected goals. Using existing VML 2.0 sequencing capabilities, we define a general pattern to the language to enable goal achievement with flexible and robust execution. Specifically, the language pattern consists of defining hierarchies, preconditions and effects familiar to the AI planning community. To implement a hierarchy, a VML sequence for a goal or activity can spawn other VML sequences for sub-activities, eventually breaking down to executable commands. When appropriate, sequence execution can be delayed to wait for the preconditions to be met, allowing more flexible execution. Finally, effects of the commands are monitored and appropriate responses can be defined to recover from failures and provide more robust execution.

Autonomous Spacecraft Operations

Our work was motivated by scenarios taken from the Autonomous Sciencecraft Experiment (ASE) used in operating the Earth-Observing 1 (EO-1) satellite. In these scenarios, the science team starts by providing a set of data collect requests that oversubscribe spacecraft resources. A baseline set of collects and alternates are selected and uplinked. During execution, on-board science processing may generate new goal requests [Chien et al., April 2005]. Ground-based sensorweb processing may do the same using uplinked commands [Chien et al., June 2005]. A prototype Goal Manager was implemented and tested on these scenarios. The EO-1 model consists of VML sequences that implement activities for operating EO-1, including collecting and downlinking science data. The system was run on a typical EO-1 collect-downlink cycle where on-board resources (e.g. science data storage) are oversubscribed. At runtime, a simple spacecraft simulator was used to mimic command behavior, including effects on resources. Goal request changes were simulated using a time-tagged file containing the change specifications.

We also studied planning and sequencing problems from the Mars Reconnaissance Orbiter (MRO) and the Mars Exploration Rover (MER) missions. From these, we identified several scenarios that might benefit from this technology. For example, in data relay scenarios an anomaly can either trigger a request for an emergency relay communication goal, or create a relay opportunity from a failed goal that releases resources. In addition, goal selection could be used to maximize the use of onboard data storage. Rejected science goals, which at first seem to oversubscribe this highly-contended resource, could be selected at runtime if more data storage is available than originally expected.

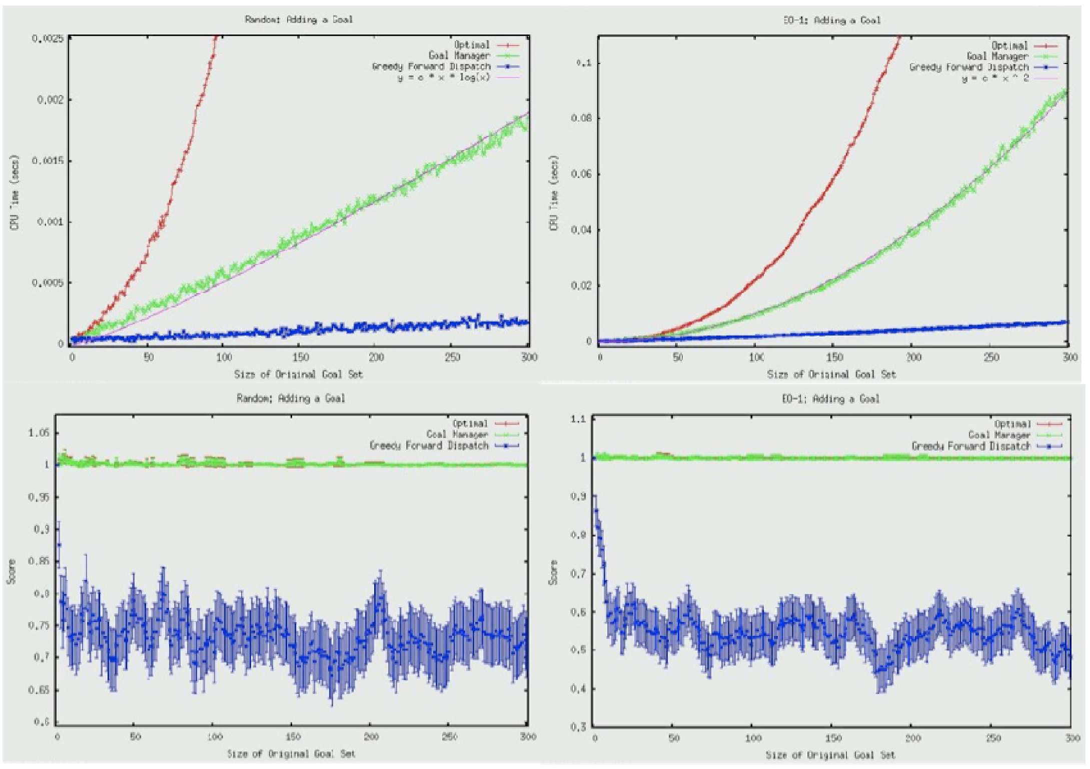

Figure 3: Empirical analyses for random and EO-1 problems.

Empirical Analysis

The motivation for our experiments is to:

- Quantitatively present the difference between our solutions and optimal solutions.
- Quantitatively present the difference between our solutions and solutions generated by a greedy, forward dispatch algorithm.
- Empirically show that the run-time of the algorithm matches the theoretical analysis.

Experiments were run on two problem sets:

- Randomly generated problems for randomly generated domains
- Randomly generated problems for the EO-1 domain

For the randomly generated domains, we identified several domain parameters that affect performance, including: number of goal types (G), number of resource types (R), and number of resources per goal (RPG). These parameters impact the level of interaction between goals. We expect higher levels of interaction to result in slower runtimes, but a higher potential for quality improvements when compared to greedy solutions.

When generating random problem sets for either a random domain or the EO-1 domain, we looked at the following parameters: number of goal instances (N), number of goals per goal type (NPG), number of goals per time unit (NPT), and number of goals per priority level (NPP). Again, the first three parameters impact the level of goal interaction. More goals in a smaller time range will have more shared resources that interfere. The last parameter, number of goals per priority level (NPP), was used to show that our solutions are optimal when guaranteed (NPP = 1) and do not stray far from optimal otherwise (NPP > 1).

Experiments were generated using a random problem generator. Specifically, random domains include 20 resource types and 20 goal types with each goal type using 10 resources. The EO-1 domain contains 7 resource types and 5 goal types with each goal using about 3 resources on average. Goals are defined for collecting, processing, and downlinking data. Shared resources are defined for the onboard data recorder and file system. For both random and EO-1 domains, we generated 300 problems sets containing 300 goals each.

Three algorithms were used to generate solutions from the problems sets:

- Goal Manager (GM)
- Optimal (OPT)
- Greedy Forward Dispatch (GFD)

GM runs the algorithm described in this paper. OPT is a variant of this algorithm that tries all possible orderings for adding goals with equal priority. This is meant to find better solutions that may have been missed by GM which breaks ties using a first-come-first-served rule. In GFD, goals with earlier requested start times are selected first, ignoring the interactions with unselected goal requests. This simplifies the resource reasoning by removing the need to propagate resource values into the future. Also, rejected goals are not maintained for future consideration, reducing the number of interactions that must be analyzed. GFD is used as a representative algorithm that is simple and fast, but more naïve at selecting goals, giving a lower bound on both runtime and solution quality. All algorithms were implemented in C++ and run on a Sun workstation configured with two 2.6 GHz AMD Opteron™ 252 processors, 16 GBytes of RAM, and the 64-bit Red Hat Enterprise Linux operating system.

The graphs show the average runtimes (Figure 3 bottom) and scores (Figure 3 top) for adding a single goal to an increasing baseline goal set for the EO-1 domain (Figure 3 right) and random domains (Figure 3 left) . The CPU times reported are for adding one new goal to a set of N goals plotted on the x-axis. The scores reported are for the resulting set of selected goals. The score of a single goal is calculated using the function $GPP_{max}^{(p - Pmax)}$ where GPP_{max} is the maximum goals per priority level, p is the priority of the goal, and P_{max} is the maximum priority of all goals. The score for a set of goals is the sum of scores for all goals in the set. This function ensures that the score for any number of goals at a lower priority will not sum up to more than the score of a single goal at a higher priority.

In these runs, start times and priorities were chosen at random from an increasing range of values, representing typical scenarios where NPT and NPP are relatively constant. This corresponds to the average case complexity analysis, and the data for random domains support the $\Theta(NlgN)$ result. The $NlgN$ fit to the GM data is shown in the graph. The EO-1 domain demonstrates the worst case where nearly all goals share the same resource (the onboard file system), and we see a best fit to an N^2 curve. The exponential runtime of OPT is due to our naïve implementation that considers all possible ordering of goals with equal priority. The graphs also show GM producing solutions with scores at or slightly lower than OPT (plotted on top of each other), but with GM and OPT both scoring much higher than GFD. The 95% confidence intervals (using standard error of the mean) are shown, but are very small for OPT and GM. The scores produced by GFD are noisy due to its preference for selecting goals with earlier start times instead of higher priority.

In other experiments we looked at performance on problems that added larger sets of goal requests. We also examined algorithm behavior along individual dimensions such RPG, NPT, and NPP. The details of these experiments are beyond the scope of this paper, but preliminary results support the theoretical analysis.

In summary, we have empirically shown our goal selection algorithm to:

- Exhibit low-order polynomial runtime behavior with respect to the size of the problem
- Generate solutions near optimal and much better than a greedy approach

In other words, with a few restrictions, we show that it is possible to preserve alternative goal sets and (when the need arises) re-consider previously rejected goals in a timely fashion in order to maintain high quality solutions.

Related Work

Much of the research in planning and goal selection has focused on more general, intractable problems. For example, [Smith 2004] looks at the more general problem that includes selecting goals and choosing their order when resource usage depends on the order of the goals (e.g., for a traveling rover). Both the Squeaky Wheel Optimization [Joslin and Clements 1999] and the Task Swapping [Kramer and Smith 2004] algorithms have been shown to improve oversubscribed schedules by re-scheduling tasks to allow more goals to fit. Instead, we look at a more constrained problem that can be solved in polynomial time, while still providing advanced autonomy capabilities useful for many embedded applications.

Tractable planning solutions typically take one of the following three approaches:

1) focus on average case performance
2) use domain-specific knowledge to simplify the general problem
3) apply general restrictions to the problem to make planning tractable

In the first approach average case performance is considered for difficult applications when occasional failures are acceptable (e.g. with the use of heuristics [Bonet and Geffner 2001]). When relying on average-case performance, one must accept the fact that the algorithm may not efficiently solve some problems. In the second approach, domain-specific knowledge is used to encode problem-specific solutions when possible including the use of hierarchies [Tate et al. 1994, Erol et al. 1994, Nau et al. 2003] or context-dependent effects [Wilkins and DesJardins 2001]. A knowledge-based solution, however, is a one that is tailored for a particular problem and can be difficult to formally verify. Our work is most closely related to the third approach, which is supported by theoretical work showing how the efficiency of planning is related to the expressivity of the planning domain language. For example, [Bylander 1994] and [Erol et al. 1995] examine limited forms of STRIPS-style operators and the effect on planning complexity. [Erol et al. 1994] investigates restricted HTN planning. [Jonsson and Backstrom 1998] examine how structural restrictions on state transition graphs impact planning complexity for the SAS+ formalism (a state variable representation). In the goal selection problem we describe, goals have fixed start times and durations. In all cases, different levels of restrictions result in different guarantees on the worst case performance of planning. For applications that can meet sufficient restrictions, planning becomes verifiably tractable.

A considerable amount of work has been done in the area of online planning and execution. We list some of the implemented systems here. For example, SCL [ICS] provides a procedural language for spacecraft commanding similar to VML. ESL [Gat 1996] is an execution language for autonomous agents, implemented as an extension to the Common Lisp programming language. TDL [Simmons et al.] extends the C++ programming language to include the concept of a task. Like ESL, programs in TDL can take advantage of the generic language on which they are based. The tradeoff, however, is that it can be much more difficult to verify programs written in an expressive language. Model-based executives such as Titan [Williams et al. 2003] and Kirk [Kim et al. 2001] use a declarative specification of system behavior (plant model) to track system state and compute desired sequences of control actions. The focus of these executives is primarily on the execution and monitoring of goals, while ours is on the selection of goals prior to execution.

Finally, our goals and resources are similar to the concepts of goals and state variables that are central to JPL's Mission Data System (MDS). MDS [Dvorak et al. 1999, Barrett et al. 2004] is a comprehensive approach to systems engineering and a methodology for the design and development of control system applications. Goal selection is a core capability required by many such applications.

Conclusions

We have described a carefully constrained set of resource and priority reasoning capabilities designed to enable run-time goal selection within a limited computational environment. These capabilities enable fast re-optimization of goal sets which oversubscribe available resources and have a strict priority ranking. We have presented both a theoretical and an empirical analysis of our algorithm as well as described its application to a number of typical spacecraft operations scenarios.

Acknowledgements

This work was performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration. A special thanks to Chris Grasso for VML software and support.

References

A A. Barrett, R. Knight, R. Morris, R. Rasmussen, Mission Planning and Execution Within the Mission Data System, *Intl Workshop on Planning and Scheduling for Space*, Darmstadt, Germany, June 2004.

B. Bonet and H. Geffner, Planning As Heuristic Search, Artificial Intelligence 129 (2001) 5-33.

T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1994) 165-204.

S. Chien, et al., Using Autonomy Flight Software to Improve Science Return on Earth Observing One, *Journal of Aerospace Computing, Information, and Communication*, April 2005.

S. Chien, et al., An Autonomous Earth-Observing Sensorweb, *IEEE Intelligent Systems*, May/June 2005.

D. Dvorak, R. Rasmussen, G. Reeves, and A. Sacks. Software architecture themes in JPL's Mission Data System. *In Proceedings of the AIAA Guidance, Navigation, and Control Conference*, number AIAA-99-4553, 1999.

K. Erol, J. Hendler, and D.S. Nau. 1994. UMCP: A sound and complete procedure for hierarchical task network planning. Proc Intl Conf AI Planning Systems.

K. Erol, D. Nau, V. Subrahmanian, Complexity, decidability and undecidability results for domain-independent planning, Artificial Int. 76 (1995) 75-88.

E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. *AAAI Fall Symp: Issues in Plan Execution*, Cambridge, MA, 1996.

Goddard Space Flight Center, The Earth Observing One Mission Page, eo1.gsfc.nasa.gov.

C. Grasso, P. Lock. VML Sequencing: Growing Capabilities over Multiple Missions. *In Proceedings of SpaceOps 08*. Heidelberg, Germany. May 2008.

Interface and Control Systems (ICS), Inc., http://www.interfacecontrol.com

P. Jonsson, C. Bäckström. State-Variable Planning Under Structural Restrictions: Algorithms and Complexity. Artificial Intelligence 100 (1998) 125-176.

D. E. Joslin and D. P. Clements, "Squeaky Wheel" Optimization, *Journal of Artificial Intelligence Research* (1999), 10:353-373.

P. Kim, B. Williams, and M. Abramson. Executing reactive, model-based programs through graph-based temporal planning. *Procs Intl Joint Conf Art Intell*, 2001.

Kramer, L. A., and Smith, S. F., Task swapping for schedule improvement, a broader analysis. *In Proc. 14th Int'l Conf. on Automated Planning and Scheduling*, 2004.

D. Nau, et al.. SHOP2: An HTN planning system. Journal of Artificial Intelligence Research, 20 (2003) 379-404.

G. Rabideau and S. Chien, Runtime Goal Selection with Oversubscribed Resources, *Procs Intl Joint Conf Art Intell (ICAPS), Workshop on Oversubscribed Planning*, Sydney, Australia, September 2008.

R. Simmons and D. Apfelbaum. TDL Quick-Reference Manual (v1.3.2). http://www-2.cs.cmu.edu/~tdl/tdl.html, 2002.

D.E. Smith, Choosing Objectives in Over-Subscription Planning, In Proc. 14th Int'l Conf. on Automated Planning and Scheduling, 2004.

A. Tate, B. Drabble, and R.B. Kirby. 1994. O-Plan2: an open architecture for command, planning, and control. In Fox, M., and Zweben, M., eds., Intelligent Scheduling. Morgan Kaufmann Pub, San Francisco, CA. 213–239.

D. Wilkins and Marie DesJardins, A Call for Knowledge-based Planning, AI Magazine, Volume 22, #1, pp. 99-115, Spring 2001.

B.C. Williams, M.D. Ingham, S.H. Chung, and P.H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *In Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, Vol. 91, No. 1, Jan. 2003, pp. 212-237.